Existential Abstractions for Distributed Reactive Systems via Syntactic Transformations

Vijay D'Silva
ETH Zurich, Switzerland
vdsilva@inf.ethz.ch

Sampada Sonalkar
Columbia University, U.S.A.
ss3119@columbia.edu

S. Ramesh
GM R&D India Science Lab,
Bangalore, India
ramesh.s@gm.com

ABSTRACT

Synchronous languages are well suited to implementation and verification of reactive systems. Large reactive systems tend to be distributed to cope with scalability and application specific demands. We propose abstractions for distributed reactive systems modelled as a set of synchronous nodes with asynchronous communication between them. The special features of synchronous programs allow us to obtain abstractions that are also valid synchronous programs only by syntactic transformations. For a given program, the set of all such abstractions forms a semi-lattice with the original program as the bottom and the most abstract program as the top element. The transformation we define is a natural basis for constructing an abstraction-refinement framework for verification. Given a program and a safety property, the abstraction-refinement process is a search in a lattice of programs obtained via syntactic transformations. We have implemented this abstraction refinement framework in a prototype tool and report our case studies.

Categories and Subject Descriptors

F.3 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Theory, Verification

1. INTRODUCTION

Reactive systems are ubiquitous and often safety critical. Synchronous programming provides a powerful abstraction for such systems based on ideas from control theory and digital circuit design. Control dominated reactive systems are designed using languages like Esterel, Statecharts, SyncCharts or Argos [4]. Large reactive systems, such as the network of processors in an automobile, tend to be distributed, consisting of synchronous nodes communicating using asynchronous mechanisms based on message passing or shared memory. Much research has recently been devoted to extending the synchronous languages to encompass a special class of distributed reactive systems called *Globally Asynchronous Locally Synchronous* (GALS) systems [6, 15].

We address the problem of efficiently verifying distributed reactive systems. As multiple proposals exist to extend the synchronous paradigm to incorporate distributed reactive systems, we abstract away the details and model all asynchronous interaction as communication à la Communicating Sequential Processes (CSP) [6]. A survey of the state-of-the-art in automated verification reveals that abstraction techniques are routinely used to cope with the prohibitive size and complexity of designs to be verified. *Localisation reduction* abstracts circuits by removing invisible latches and their associated logic and using spurious counterexamples to refine the abstraction [3, 12]. This technique operates on a low level synchronous design and may not be applicable to distributed reactive systems. In *predicate abstraction* [16], abstract programs are constructed using Boolean variables tracking predicates on data values. Unlike software, the behaviour of control dominated distributed reactive systems is dictated by volatile signals, events and asynchronous communication. We propose specialised abstractions for distributed reactive systems that exploit semantic aspects of the synchronous model. For the rest of the paper, we use the terms program and system for synchronous programs and distributed reactive systems respectively.

Verification algorithms for safety properties usually construct *existential abstractions*. Consider two synchronous programs M_1 and M_2 with the same signals. If every execution sequence in M_1 has a corresponding execution sequence in M_2 such that the temporal behaviour of M_2 is identical to that of M_1 on some signals but arbitrary on others, M_2 is an existential abstraction of M_1. Conversely, we say that M_1 refines M_2. Any safety property that holds in M_2 holds in M_1 as well. We prove that the core synchronous programming constructs of parallel and hierarchical composition give rise to such refinements and provide modular refinement results for four basic operators. Signal localisation and the interaction between synchronous and asynchronous behaviour introduce subtle complications which require consideration of the program's context.

The theoretical results we prove are used to construct abstractions by syntactically modifying a program. These syntactic transformations have two desirable features: (1) expensive calls to a theorem prover are avoided when constructing abstractions and (2) the abstraction is a valid synchronous program, which can be analysed using standard

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

Copyright 2007 ACM 978-1-59593-825-1/07/0009 ...$5.00.

compilers and verification tools. Additionally, we show that the set of all such syntactically constructed abstractions forms a semi-lattice with the original program as the bottom element and the most abstract program as the top. Counterexample guided abstraction refinement [7] for the systems in this paper can be viewed as a search in this lattice for the optimal abstraction.

We have implemented our verification algorithm in a prototype tool for graphically describing and simulating distributed reactive systems [15]. A concrete model is first abstracted by dropping components which do not generate signals of interest. The abstract model is verified in a model checker. Counterexamples from the model checker are translated into traces that can be run in the simulator to determine if they are feasible. Spurious traces are used to refine the abstraction by adding a component that may previously have been eliminated. We have conducted several case studies and in many cases, have observed a significant reduction in the number of Boolean state variables the model-checker had to consider in many cases.

The paper is organised as follows: Section 2 introduces our main idea informally, through an example. In Section 3, we formally define the model and in Section 4, the basis of our abstraction technique. Section 5 covers our case-studies, Section 6 discusses related work and concludes.

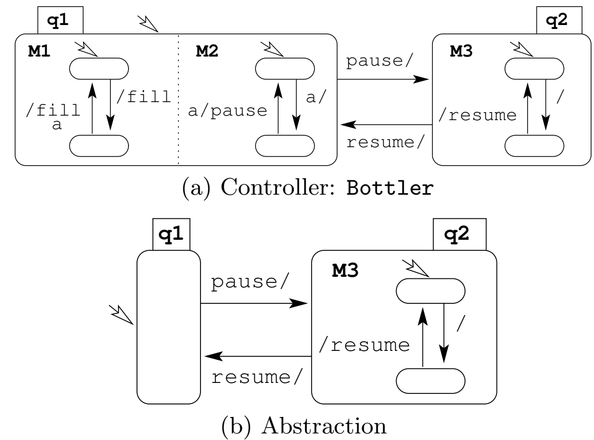
2. OVERVIEW BY EXAMPLE

We illustrate our ideas with an example of a controller. We begin with a synchronous controller and then consider a distributed controller system.

Consider a bottling machine in a factory. Vertically oriented bottles arrive on a conveyor belt and are filled, four at a time. The four bottles are then moved out from under the nozzles and are capped. Capped bottles are lifted off the conveyor belt and placed in a crate. A crate can hold sixteen bottles. When a crate is full, it is moved elsewhere. To prevent a pile-up on the conveyor belt, bottles are not filled while a crate is being moved. Replacing a full crate with an empty one takes twice the amount of time required to fill four bottles. After a new crate is in place, the sequence of operations above repeats.

A synchronous controller `Bottler` for the bottling machine is shown in Figure 1(a). At the top level, it is a *reactive automaton* (RA) called `Mode` with two states q_1 and q_2 as shown (the label `Mode` is not shown in the figure to avoid clutter). A reactive automaton executes in discrete steps, each identified with a *time instant*. In each step, depending on the input signals and the current state, a transition is made to the next state and output signals are simultaneously emitted. Bottles are filled in state q_1 but not in state q_2. A transition is made from q_1 to q_2 when a `pause` signal is received. A transition is made from q_2 to q_1 when the signal `resume` is received.

The state q_1 is *hierarchical* and contains the reactive automata $\mathtt{M}_1$ and $\mathtt{M}_2$. $\mathtt{M}_1$ generates the signal `fill`, which causes the machine to fill bottles. If the `fill` signal has been generated four consecutive times, the automaton $\mathtt{M}_2$ generates the signal `pause`, which causes the transition from q_1 to q_2. The state q_2 is also hierarchical, containing the reactive automaton $\mathtt{M}_3$. The controller remains in q_2 when a crate is being replaced. Two time instants after entering q_2, $\mathtt{M}_3$ generates the signal `resume`, which causes a transition to q_1 after which the machine continues filling bottles.



(a) Controller: `Bottler`

(b) Abstraction

Figure 1: Bottling machine controller and abstraction

The RAs $\mathtt{M}_1$ and $\mathtt{M}_2$ in `Bottler` are said to be in *synchronous parallel composition*, denoted $\mathtt{M}_1 \| \mathtt{M}_2$. A hierarchical state q containing an RA M is denoted $[q \to M]$. The RA `Bottler` can be written textually as $\mathtt{Mode}[q_1 \to (\mathtt{M}_1 \| \mathtt{M}_2)][q_2 \to \mathtt{M}_3]$.

Suppose we want to verify the safety property that two steps after receiving a pause signal, the controller resumes operation. The entire behaviour of the controller need not be analysed to determine that this property holds. It may be sufficient to prove this property for a sound abstraction of the RA `Bottler`. We use synchronous programs as abstractions. An abstraction which suffices is the RA shown in Figure 1(b). Observe that this abstraction is obtained by simply dropping the hierarchical components from the state q_1. The RA is an abstraction because the signals `fill` and `pause` are treated as inputs. Their behaviour, being undefined may be arbitrary. Thus, the traces of the abstraction are a superset of those of `Bottler`.

We prove that sound abstractions can be obtained by dropping parallel and hierarchical components of reactive automata. We also show how abstractions can be constructed in the presence of signal localisation and asynchronous communication. The set of abstractions constructed in this manner forms a lattice. This lattice of syntactically constructed abstractions for `Bottler` is shown in Figure 2.

We now introduce an asynchronous node in this system. A second controller may generate an interrupt that causes the bottling machine to abort. This controller, modelled by the RA `Int`, is shown in Fig 3(b) along with an extension of `Bottler`, called `Aborttler`, that allows for operation to be aborted when the machine is not filling bottles. States in which asynchronous communication can take place are drawn as ellipses for emphasis and the relevant transitions are drawn dashed. There is an asynchronous channel `ab` connecting the two controllers. The CSP communication action `ab`? succeeds iff the action `ab`! occurs simultaneously. If no asynchronous communication takes place, either one controller or the other executes, as is standard in most asynchronous concurrent formalisms. The entire system, is a *communicating reactive machine* CRM; the asynchronous (or interleaved) composition of the two controllers is denoted `Aborttler` $|||$ `Int`.

In the presence of asynchronous communication, reactive automata cannot be simplified as before to obtain existential

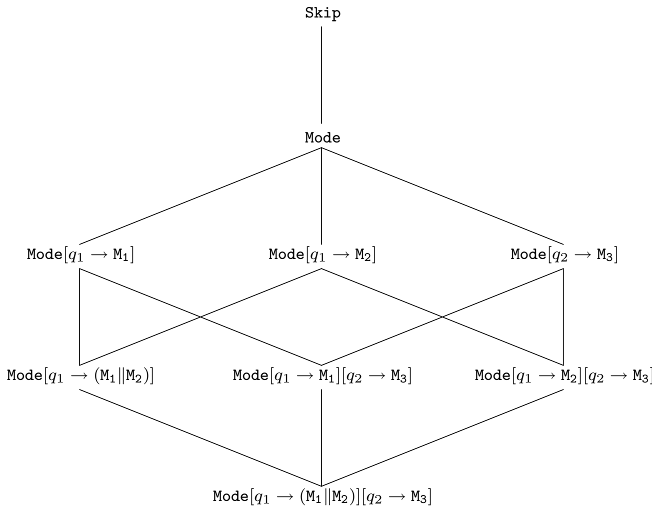
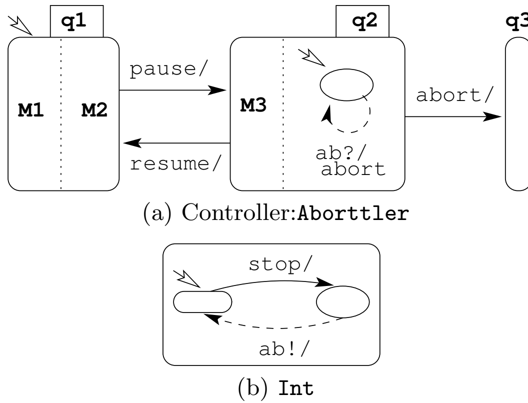


Figure 2: Lattice of Abstractions



(a) Controller:**Aborttler**

(b) **Int**

Figure 3: Bottling and Interrupt Controllers

abstractions. If an automaton is involved in asynchronous communication, removing it might lead to a deadlock in the transformed system. If this happens, the transformed system will have fewer traces than the original and cannot be used as an abstraction. In Section 4 we identify conditions which must be satisfied to obtain abstractions in the presence of asynchronous communication.

3. THE FORMAL MODEL

We introduce a simple process calculus that contains the main synchronous programming constructs, along with asynchronous communication. This is along the lines of the Communicating Reactive Processes (CRP) model for a network of Esterel reactive programs presented in [6].

Definition 1. The syntax of a communicating reactive machine (CRM) is defined by the grammar:

$$
\begin{array}{lll}
\text{CRM} & ::= & \text{RA} \mid \text{CRM} \,|||\, \text{CRM} \\
\text{RA} & ::= & \text{HA} \mid \text{RA} \| \text{RA} \mid \text{RA} \backslash s \\
\text{HA} & ::= & \text{AUT} \mid \text{HA}[q \to \text{RA}] \\
\text{AUT} & ::= & (Q, init, \mathcal{I}, \mathcal{O}, \mathcal{C}, \mathcal{T})
\end{array}
$$

where AUT is a deterministic Mealy-style automaton with input $\mathcal{I}$, output $\mathcal{O}$ and channels $\mathcal{C}$. For a set S of signals, let $\mathcal{B}(S)$ denote the set of Boolean formulae over S.

$$
\frac{K_1 \xrightarrow[e_1]{o_1, c_1} K_1' \;,\; c_1 \cap \mathcal{C}_2 = \emptyset}{K_1 \,|||\, K_2 \xrightarrow[e_1]{o_1, c_1} K_1' \,|||\, K_2} \quad [\texttt{Asynch1}]
$$

$$
\frac{K_2 \xrightarrow[e_2]{o_2, c_2} K_2' \;,\; c_2 \cap \mathcal{C}_1 = \emptyset}{K_1 \,|||\, K_2 \xrightarrow[e_2]{o_2, c_2} K_1 \,|||\, K_2'} \quad [\texttt{Asynch2}]
$$

$$
\frac{K_1 \xrightarrow[e_1]{o_1, c_1} K_1' \;,\; K_2 \xrightarrow[e_2]{o_2, c_2} K_2', c_1 \cap \mathcal{C}_2 = c_2 \cap \mathcal{C}_1}{K_1 \,|||\, K_2 \xrightarrow[e_1 \wedge e_2]{o_1 \cup o_2, c_1 \cup c_2} K_1' \,|||\, K_2'} \quad [\texttt{Asynch3}]
$$

Table 1: Asynchronous Transition rules

Let $A^{\mathcal{O}} = 2^{\mathcal{O}}$ denote the set of output actions; sets of signals that may be emitted in a transition. For a channel c, $c!$ is a send action and $c?$ is a receive. Let $A^{\mathcal{C}}$ denote the set of asynchronous actions such that $c? \in A^{\mathcal{C}}$ iff $c! \notin A^{\mathcal{C}}$ and vice versa. The transition relation $\mathcal{T} \subseteq \big(Q \times \mathcal{B}(\mathcal{I} \cup \mathcal{O}) \times A^{\mathcal{O}} \times Q\big) \cup \big(Q \times A^{\mathcal{C}} \times A^{\mathcal{O}} \times Q\big)$ contains synchronous and asynchronous transitions.

A communicating reactive machine is the asynchronous composition of a set of reactive automata. A reactive automaton M may be in synchronous parallel composition with another, denoted $M \| N$, may have a hierarchical state q containing an RA N, denoted as an HA, $M[q \to N]$ and may have local signals, denoted $(M \backslash s)$ for each local signal s. Local signals have limited scope. For example, in the RA $(M \backslash s) \| N$, the signal s is not visible in the RA N. An RA with no structuring constructs is simply an automaton (AUT). An automaton has states and transitions. In a synchronous transition, if a condition on the input signals is satisfied, an set of output signals is emitted. In an asynchronous transition, if communication succeeds on a channel, a set of output signals is emitted. We briefly define the semantics of our model, which is based on Esterel [5].

3.1 Semantics

An event e is a minterm over the complete set of signals of an RA. It is a conjunction of positive and negative literals indicating presence and absence of signals. Events and asynchronous communication define which transitions take place. At any step in the execution of an RA, we say that the current state is *active*. The semantics of a CRM M is given by a sequence of configurations of the form $K_i = (M, A_i)$ where A_i is a set of active states in the RAs in M at time step $i \in \mathbb{N}$; essentially the synchronous program counter. Transitions are uniformly denoted as $(M, A_i) \xrightarrow[e]{o,c} (M, A_{i+1})$. In a synchronous transition, the states in A_i are active and the event e causes signals o to be emitted and c is the empty set. In an asynchronous transition, communication takes place on the channels in c and signals in the set o are emitted. For an asynchronous action a being $c?$ or $c!$, define $chan(a) = \{c\}$. Let $init(M)$ denote the set of initial states of all the RAs in M.

Table 1 describes the transitions of an asynchronous system $M_1 \,|||\, M_2$ in configurations K_1 and K_2, with sets of channels $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. These rules are similar to those for parallel composition in CSP with the important difference that communication on multiple channels is possible in a single transition. If no communication occurs, either M_1 or M_2 makes a transition (rules `Asynch1` or `Asynch2`). If communication takes place, M_1 and M_2 progress together

(rule `Asynch3`). Note that multiple communications can take place in the same transition step.

Table 2 describes the semantics of synchronous constructs. Synchronous transitions in a simple automaton are described by the rule `Synch`. If the state q is active, a transition from q to q' is taken provided the input condition and output signals are consistent with the event e. Asynchronous transitions are described by `Asynch`, which is similar.

Two RAs in synchronous parallel composition have the same signals. In the rule `SynchPar`, a pair of transitions in the two RAs, which are consistent with the set of active states and a given event can take place simultaneously. In contrast, note that in an asynchronous composition, the events of the two RAs are different.

The semantics of hierarchical states are given by three rules. In a transition leaving a hierarchical state in rule `Hsrc`, the RAs contained in the state first complete their execution. This is known as *weak preemption* and is standard in the synchronous languages. The active states after the transition are only A'_1 and not $A'_1 \cup A'_2$. For example, in the transition from q_1 to q_2 in Figure 1(a), the RA $\mathtt{M}_1$ first generates the signal `fill`. If a hierarchical state q containing the RA M_2 is entered, as described in the rule `Htarget`, the set of active states after the transition includes q and the initial states of M_2. An example is the transition from q_1 to q_2 in Figure 1(a), in which the set of active states after the transition contains q_2 and the initial states of $\mathtt{M}_3$. Finally, in the rule `Hself` describing a transition from a hierarchical state to itself, all contained RAs complete their current transitions and are then "reset" to their initial states.

Signal localisation is a well studied issue in the synchronous languages because of causality paradoxes which may arise. We adopt Berry's constructive solution and explain it briefly. A comprehensive discussion can be found in [5]. A signal s can only be made local if its status can be uniquely defined. This means that assuming the status of s undefined, denoted $s^{\perp}$, we can determine if at the end of the reaction, s is present or absent, denoted s^{+} and s^{-} respectively, by examining how the other signals affect the output. Let $e * s^{\perp}$ denote the expression obtained by replacing every occurrence of s by $s^{\perp}$ in e. Let $must(K, e * s^{\perp})$ denote the set of signals that are emitted in every transition from the configuration K with the event $e * s^{\perp}$. As the event e is not completely defined, multiple transitions may be possible. In the rule `Local1`, s is in the *must* set, so it will always be emitted and we can assume that it is present in e to compute the next states, but have to remove it from the output set because it is local. Let $may(K, e * s^{\perp})$ denote the set of signals that are emitted in at least one transition from the configuration K with the event $e * s^{\perp}$. Note that $must(K, e * s^{\perp}) \subseteq may(K, e * s^{\perp})$. In rule `Local2`, s is *not* in the *may* set, so it will never be emitted in transitions from that configuration. We can assume that s is absent to compute the next set of active states. In both rules, the status of the signal can be determined without making any a priori assumptions. That is, without cyclic reasoning. If the signal is not in the *must* set but is in the *may* set, it may be emitted but need not be. Such programs are rejected by compilers of synchronous languages because their behaviour is unpredictable.

A *run* of a CRM M is sequence of transitions derived from the semantic rules, $K_0 \xrightarrow[e_0]{o_0, c_0} K_1 \xrightarrow[e_1]{o_1, c_1} \ldots$, where the initial configuration $K_0 = (M, init(M))$. A *trace* τ is a sequence $(e_0, o_0, c_0), (e_1, o_1, c_1), \ldots$ derived from a run. We write e_i, o_i and c_i as $\tau[i].e$,$\tau[i].o$ and $\tau[i].c$ respectively. Let $Tr(M)$ denote the set of all traces of M.

4. EXISTENTIAL ABSTRACTION

We are interested in existential abstractions. Existential abstractions are typically defined such that for every trace in the concrete program, there exists a corresponding trace in the abstraction. Any safety property that is satisfied by an existential abstraction also holds for the concrete program. The mathematical relation between an abstraction and a concrete system is called refinement.

We define refinement for CRMs based on these ideas. Our definition may appear different from existing definitions because our model combines both synchrony and asynchrony. In a synchronous model, if an RA N emits more output signals than RA M for the same sequence of events, the behaviour of N is more defined than that of M because we can make stronger statements about N. The most abstract RA has all signals as inputs. Thus, all signals behave completely arbitrarily and nothing can be proved about them. If a signal that N produces as output is made local, it is not externally visible in the trace anymore but may influence the set of traces of N. Hence, local signals have to be treated specially for abstraction.

In contrast to output signals, asynchronous communication actions should increase in an abstraction. This is because more asynchronous communication leads to more traces in a CRM. If fewer communication actions take place, one of the asynchronous nodes may deadlock, leading to a smaller set of traces in the entire system. The most abstract CRM is like Hoare's `Chaos` process, arbitrarily communicating on all possible channels. Definition 2 is motivated by these observations.

Definition 2. A CRM N refines M in a context with signals S, denoted $N\ \mathtt{ref}^S\ M$, iff for every $\tau \in Tr(N)$, there exists a trace $\sigma \in Tr(M)$ such that at each step i the following holds:

1. $\tau[i].e = \sigma[i].e$ and $\tau[i].o \supseteq \sigma[i].o$ and $\tau[i].c \subseteq \sigma[i].c$
2. $\tau[i].o \cap S = \sigma[i].o \cap S$

The set S is a parameter that is used when dealing with local signals and to construct the initial abstraction. We write $N\ \mathtt{ref}\ M$ for $N\ \mathtt{ref}^{\emptyset}\ M$. If $S' \subseteq S$, then $N\ \mathtt{ref}^S\ M$ implies $N\ \mathtt{ref}^{S'}\ M$.

4.1 Syntactic Transformations

We now define transformations of CRMs and show that they give rise to existential abstractions. First, we consider RAs constructed using only synchronous parallel composition and hierarchy. Let `Skip` denote the reactive automaton with all signals as input and no asynchronous communication. For an RA M, let $Components(M)$ denote the set of RAs constituting M. Let A denote an automaton. The set $Components(M)$, abbreviated to $Com(M)$ is formally defined as follows:

$$\frac{(q,i,o,q') \in T_M, e \to i \wedge o}{(M,\{q\}) \xrightarrow[e]{o,\emptyset} (M,\{q'\})} \quad \texttt{[Synch]} \qquad \frac{(q,a,o,q') \in T_M}{(M,\{q\}) \xrightarrow[e]{o,chan(a)} (M,\{q'\})} \quad \texttt{[Asynch]}$$

$$\frac{K_1 \xrightarrow[e]{o_1,c_1} K_1', K_2 \xrightarrow[e]{o_2,c_2} K_2'}{K_1 \| K_2 \xrightarrow[e]{o_1 \cup o_2, c_1 \cup c_2} K_1' \| K_2'} \quad \texttt{[SynchPar]}$$

$$\frac{(M_1,A_1) \xrightarrow[e]{o_1,c_1} (M_1,A_1'), (M_2,A_2) \xrightarrow[e]{o_2,c_2} (M_2,A_2'), q \in A_1, q \notin A_1'}{(M_1[q \to M_2], A_1 \cup A_2) \xrightarrow[e]{o_1 \cup o_2, c_1 \cup c_2} (M_1[q \to M_2], A_1')} \quad \texttt{[Hsrc]}$$

$$\frac{(M_1,A_1) \xrightarrow[e]{o_1,c_1} (M_1,A_1'), q \notin A_1, q \in A_1'}{(M_1[q \to M_2], A_1) \xrightarrow[e]{o_1,c_1} (M_1[q \to M_2], A_1' \cup init(M_2))} \quad \texttt{[Htarget]}$$

$$\frac{(M_1,A_1) \xrightarrow[e]{o_1,c_1} (M_1,A_1'), (M_2,A_2) \xrightarrow[e]{o_2,c_2} (M_2,A_2'), q \in A_1, q \in A_1'}{(M_1[q \to M_2], A_1 \cup A_2) \xrightarrow[e]{o_1 \cup o_2, c_1 \cup c_2} (M_1[q \to M_2], A_1' \cup init(M_2))} \quad \texttt{[Hself]}$$

$$\frac{s \in must((M,A), e * s^{\perp}), (M,A) \xrightarrow[e*s^+]{o_1,c_1} (M,A')}{(M \backslash s, A) \xrightarrow[e]{o_1 - \{s\}, c_1} (M \backslash s, A')} \quad \texttt{[Local1]} \qquad \frac{s \notin may((M,A), e * s^{\perp}), (M,A) \xrightarrow[e*s^-]{o_1,c_1} (M,A')}{(M \backslash s, A) \xrightarrow[e]{o_1,c_1} (M \backslash s, A')} \quad \texttt{[Local2]}$$

Table 2: Synchronous Transition rules

$$\begin{aligned} Com(A) &= \{A\} \\ Com(M_1 \| M_2) &= \{M_1 \| M_2\} \cup Com(M_1) \cup Com(M_2) \\ Com(M_1[q \to M_2]) &= Com(M_1) \cup \\ &\quad \{M_1[q \to M_2'] | M_2' \in Com(M_2)\} \end{aligned}$$

Definition 3. Given an RA M, define the relation $\sqsubseteq$ between RAs $M_1, M_2 \in Components(M)$ such that if $M_2 \in Components(M_1)$ then $M_1 \sqsubseteq M_2$. For all RAs N define $N \sqsubseteq \texttt{Skip}$.

Observe that as per the definition, for two RAs M_1 and M_2, $M_1 \| M_2 \sqsubseteq M_1$ and $M_1[q \to M_2] \sqsubseteq M_1$. Similarly $M_1 \| M_2 \sqsubseteq M_2$ and $M_2[q \to M_1] \sqsubseteq M_2$.

LEMMA 1. *The relation $\sqsubseteq$ is a partial order on the set $Components(M)$, where M is an RA with only synchronous parallel and hierarchical composition.*

PROOF. We need to show that $\sqsubseteq$ is reflexive, antisymmetric and transitive. Observe that $M \in Components(M)$, so $M \sqsubseteq M$. Now observe that $M \sqsubseteq N$ iff $Components(N) \subseteq Components(M)$. Thus, if $M \sqsubseteq N$ and $N \sqsubseteq M$, we know that $Components(N) = Components(M)$. Suppose M and N are not identical. Then, there exist either an RA or composition of RAs in M that is not in N. But then, this RA should also be in $Components(M)$ and not in $Components(N)$, which cannot be. Transitivity follows because if $P \sqsubseteq Q$ and $Q \sqsubseteq R$, then $Components(R) \subseteq Components(Q) \subseteq Components(P)$, so $P \sqsubseteq R$ as required. □

Additionally, for an RA M, we can define a meet semilattice of RAs on the set $Components(M) \cup \{\texttt{Skip}\}$. The meet of $M_1, M_2 \in Components(M)$, denoted $M_1 \sqcap M_2$, is the RA N such that $Components(N)$ is the smallest subset of $Components(M)$ containing both M_1 and M_2. M is the bottom and `Skip` is the top of this lattice. Let $\mathbb{L}(M)$ denote the lattice $\langle Components(M) \cup \{\texttt{Skip}\}, \sqsubseteq, \sqcap \rangle$ for an RA M. For example, the set $Components(\texttt{Bottler})$ for the RA in Figure 1(a) is the set of nodes in Figure 2. The partial order and $\mathbb{L}(\texttt{Bottler})$ are as shown in Figure 2.

THEOREM 1. *For reactive automata $M_1, M_2 \in \mathbb{L}(M)$, if $M_1 \sqsubseteq M_2$, then M_1 `ref` M_2.*

PROOF. We know from the definition of the lattice that a sequence of RAs exists such that $M_1 = N_0 \sqsubseteq N_1 \ldots \sqsubseteq N_k \sqsubseteq N_{k+1} = M_2$ and for any $0 \leq i \leq k$, N_{i+1} is the least element in $\mathbb{L}(M)$ greater than N_i. The proof is by induction on k. For the base case, $k = 0$, we have $N_0 = M_2$. Clearly M_1 `ref` M_1.

For the induction hypothesis, assume that for any k in the sequence above that M_1 `ref` N_k. It remains to show that M_1 `ref` N_{k+1}. By definition of $\sqsubseteq$, either (1) N_{k+1} is `Skip` or (2) N_k must be of the form $N_{k+1} \| N'$ or $N_{k+1}[q \to N']$. In case (1), for every trace $\tau = (e_0, o_0, \emptyset), (e_1, o_1, \emptyset), \ldots$ of $Tr(N_k)$, there exists a trace $\sigma = (e_0', \emptyset, \emptyset), (e_1', \emptyset, \emptyset), \ldots$ in $Tr(\texttt{Skip})$ with the same sequence of events and no outputs. Thus, N_k `ref` `Skip`. In case(2), we need to consider the semantic rules for synchronous parallel and hierarchical composition. Observe in the rules `SynchPar`, `Hsrc`, `Htarget` and `Hself` that a transition is defined in the composition iff there is a transition with the same event in the two RAs composed. Further, the output signals emitted in a transition in the composition are the union of those emitted by the two RAs. Thus, for every trace of the composition, we can derive corresponding traces in the composed RAs satisfying condition 1 of Definition 2 and can conclude that N_k `ref` N_{k+1}.

Hence, it holds that M_1 `ref` N_{k+1} and that M_1 `ref` M_2. □

Let us examine the practical utility of these results. The set $Components(M)$ can be constructed entirely by syntactically modifying M. From Theorem 1 we know that these RAs are all existential abstractions of M. Synchronous composition and hierarchy are by far the most commonly used structuring operators in the synchronous languages, so we expect such abstractions can be frequently constructed. However, we have made two strong assumptions so far: that the RAs contain no signal localization and no asynchronous communication. In the next section, we show that a syntactically constructed abstraction can be combined with other RAs to obtain more general refinement results.

4.2 Modular Refinement

If we have a pair of RAs M_1 and M_2 such that $M_1 \,\mathtt{ref}\, M_2$, and M_1 is part of a larger program, it is desirable, if possible, to replace M_1 by M_2 to reduce the complexity of verification.

If M_1 is in parallel or hierarchical composition with some other RAs, it can be replaced by M_2. The intuition is similar to that used in the proof of Theorem 1. Traces in the synchronous parallel and hierarchical composition of two RAs are constructed from traces of their components and have more outputs. If $(M_1 \,\mathtt{ref}\, M_2)$ we can also conclude via similar arguments that $(M_1 \| M_3) \,\mathtt{ref}\, (M_2 \| M_3)$. A similar statement can be made for hierarchical composition. Such a result is useful because we can individually simplify different parts of a large system without having to examine the monolithic program.

We have remarked before that signal localization complicates matters. Consider an RA M and $\mathbb{L}(M)$. Suppose a signal s is made local, then s is no longer an input or output of $M \backslash s$, so we cannot compare $M \backslash s$ with any RA in $\mathbb{L}(M)$. What if we consider the RA $N \backslash s$ for some $N \in \mathbb{L}(M)$? It may be that the signal s is emitted in M but not in N, so the results of the *may-must* analysis will be different. Consequently, the transitions defined in $M \backslash s$ and $N \backslash s$ will be different, so even though $M \,\mathtt{ref}\, N$, no similar conclusion can be drawn about $M \backslash s$ and $N \backslash s$.

Table 3 contains simple examples illustrating refinement and modular refinement. The columns *Concrete* and *Abstract* contain programs related by $\mathtt{ref}^S$, with S in the last column. The abstractions in the first three rows are constructed as in the previous section. Observe that $(\mathtt{M}_1 \| \mathtt{M}_2) \mathtt{ref}$ $\mathtt{M}_2$ but $(\mathtt{M}_1 \| \mathtt{M}_2) \backslash a$ and $\mathtt{M}_2 \backslash a$ behave differently because $\mathtt{a}$ is in the *must* set of the concrete but not the abstract program. Similarly, we know from the second row that $(\mathtt{M}_1 \| \mathtt{M}_2) \,\mathtt{ref}^{\{b\}}$ $\mathtt{M}_2$ and can obtain the abstraction $\mathtt{M}_2 \| \mathtt{M}_3$ of $(\mathtt{M}_1 \| \mathtt{M}_2 \| \mathtt{M}_3)$ without examining $\mathtt{M}_3$ such that $(M_1 \| M_2 \| M_3) \,\mathtt{ref}^{\{b\}}\, (M_2 \| M_3)$. For the example in row 6, an abstraction is constructed modularly for an RA with hierarchical composition.

Let us examine abstractions for CRMs. In the CRM $M \,|||\, N$, we cannot drop one of the parallel components to obtain an existential abstraction. To see why, consider the CRMs in row 7 of Table 3. If the node $\mathtt{M}_5$ is removed, no asynchronous communication takes place, so we do not have an abstraction. A possibility is to replace an asynchronous node by a `Chaos` process that communicates arbitrarily on all channels. Though we get an abstraction, it may be far too coarse to help prove any property.

What if we consider abstractions of M and N in $M \,|||\, N$? That is, if $M = M_1 \| M_2$, what can we conclude about $M_1 \,|||\, N$ given that $M \,\mathtt{ref}^S\, M_1$? Once again, the problem is that any asynchronous communication between M_2 and N is lost. If the RA $\mathtt{M}_4$ in row 7 of Table 3 is removed, no asynchronous communication takes place, and the set of traces in the system is reduced. A remedy, is to introduce the `Chaos` process, but restrict its asynchronous behaviour to only those channels in $\mathtt{M}_4$. This way, given $M \,|||\, N$, we can pick an abstraction of M from $\mathbb{L}(M)$ and use it to obtain an abstraction of $M \,|||\, N$ without introducing too much spurious behaviour. We have the following *modular refinement* results.

THEOREM 2. *Let M_1, M_2, M_3 and N_1, N_2, N_3 be RAs.*

1. *If $M_1 \,\mathtt{ref}^S\, M_2$, then $(M_1 \| M_3) \,\mathtt{ref}^S\, (M_2 \| M_3)$.*
2. *If $M_1 \,\mathtt{ref}^S\, M_2$, then $M_3[q \to M_1] \,\mathtt{ref}^S\, M_3[q \to M_2]$.*
3. *For $s \notin S$, if $M_1 \,\mathtt{ref}^{S \cup \{s\}}\, M_2$, then $M_1 \backslash s \,\mathtt{ref}^S\, M_2 \backslash s$.*
4. *If $N_1 \,\mathtt{ref}^S\, N_2$, then $N_1 \,|||\, N_3 \,\mathtt{ref}^S\, N_2 \,|||\, N_3$.*

4.3 Constructing Abstractions

Given an arbitrary CRM, we would like to construct a sequence of abstractions using syntactic transformations as much as possible. From Section 4.1, we know that this is possible for a purely synchronous system that includes only synchronous parallel and hierarchical composition. We show how $\mathbb{L}(M)$ can be used to obtain abstractions for CRMs containing M.

For an RA M, let $chan(M)$ be the set of channels on which asynchronous communication takes place in M. Consider a CRM $N_1 \,|||\, N_2$. To construct abstractions for this system we first obtain an RA M from $\mathbb{L}(N_1)$ as follows:

1. If $N_1 = M_1 \| \ldots \| M_k$, pick an $M \in \mathbb{L}(N_1)$.
2. If $N_1 = M_0[q_1 \to M_1] \ldots [q_k \to M_k]$, pick $M \in \mathbb{L}(N_1)$.
3. If $N_1 = N \backslash s$, pick $M \in \mathbb{L}(N)$ such that $N \,\mathtt{ref}^{\{s\}}\, M$.

In the presence of asynchronous communication, Let $C = chan(N_1) \setminus chan(M)$. That is, C is the set of channels on which communication actions may not take place in the abstraction. Define $\mathtt{Chaos}(C)$ as the RA that communicates arbitrarily on the channels in the set C. We can use $\mathtt{Chaos}(C) \| M$ as an abstraction of N_1. More precisely, $N_1 \,|||\, N_2 \,\mathtt{ref}\, (\mathtt{Chaos}(C) \| M) \,|||\, N_2$. In this manner, we can construct a sequence of existential abstractions for CRMs, using abstractions from the lattice. The coarsest abstraction of N_1 is $\mathtt{Chaos}(chan(N_1)) \| \mathtt{Skip}$.

We use this scheme to construct abstractions that are refined using counterexamples. Given a safety property φ and a CRM $N_1 \,|||\, N_2$, a *minimal* abstraction for φ is the CRM $M_1 \,|||\, M_2$ where M_i is derived from $M_i' \in \mathbb{L}(N_i)$ and M_i' is the maximal element of $\mathbb{L}(N_i)$ that can be used to prove φ. Note that because of signal localization and introducing a `Chaos` process, M_i itself may not be in $\mathbb{L}(N_i)$. In general, there may not exist a *best* abstraction, that is, a unique minimal abstraction.

5. EXPERIENCE

We have described a simulator and model checker for our distributed reactive systems model in [14, 15]. We have added an abstraction-refinement algorithm to the tool based on the ideas in this paper.

The results of the previous section have been algorithmically encoded in the iterative-refinement loop in Figure 4. Specifications are provided as distributed observers, a generalisation of the standard automata based specifications used in the synchronous languages [11]. An initial abstraction is constructed retaining only components generating signals in the specification. A CRM and it's specification are translated into PROMELA for verification with SPIN. If model checking an abstract model fails, we obtain a counterexample. The counterexample is translated into a sequence of events and communication actions which can be fed as input to the simulator.

The simulator serves two purposes: To identify spurious counterexamples and to evaluate the quality of refinement,

	Concrete	Abstract	S
1	M1 /a ⋮ M2 a/b	M1 /a	$\{a\}$
2	M1 /a ⋮ M2 a/b	M2 a/b	$\{b\}$
3	M2 a/b, b/c, M3	b/c, M3	$\{c\}$
4	(M1||M2)\a /b	M1\a /	$\emptyset$
5	M1 /a ⋮ M2 a/b ⋮ b/c M3	M2 a/b ⋮ b/c M3	$\{b\}$
6	M1 /a ⋮ M2 a/b, b/c, M3	M2 a/b, b/c, M3	$\{b\}$
7	M4 r!/a, b/c, M3 ⋮ M1 a/b; M5 r?/d	M4 r!/a, b/c, M3; M5 r?/d	$\{a\}$

Table 3: Refinement: Constructs and Modularity

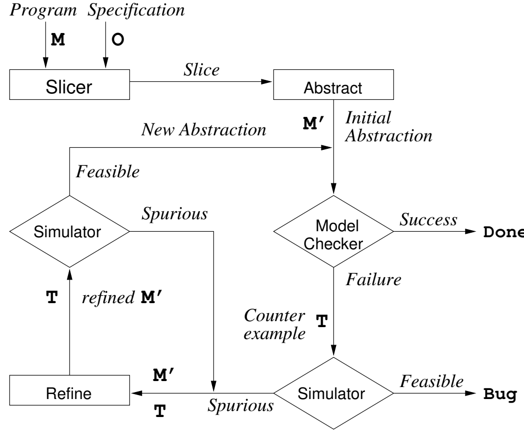


Figure 4: Iterative-Refinement loop

both roles undertaken by a theorem prover in predicate-abstraction. To determine the authenticity of the counter example, the trace is simulated on the abstract model and the original model. At each step, the simulator checks whether the set of active states in the abstract model is a subset of the set of active states in the original model. If this condition holds throughout the simulation and the observer reaches the bad state, the counter example is genuine. If at some step the subset condition does not hold, we can infer that the counter example is not a valid trace of the original model. For example, consider row 3 of Table 3 and a trace $\{a\}$. The original model enters state $\mathtt{M}_3$ and emits c while the abstract model remains in state $\mathtt{M}_1$ because signal b is absent here. On discovering a spurious counterexample, we try alternate refinements till one eliminating the spurious trace is found. The simulation step at which the subset condition failed gives a good indication that the producer of b, Component $\mathtt{M}_2$, should be added to the abstract model. The new abstraction is given as input to the model checker and the iterative verification process repeats.

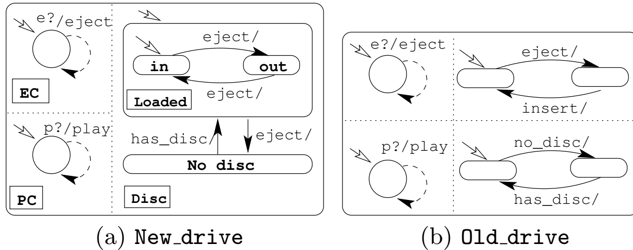
Our tool is implemented in C and is meant to be a prototype for exploring ideas for such systems. Translating a synchronous model to Promela is non-trivial and far from optimal, because the translated code has to preserve synchronous semantics where required. Thus, our running times are often quite high. However, our aim is to evaluate the efficacy of the abstraction procedure suggested here. For this, we believe it is sufficient to examine the reduction in running time and states explored during verification when using iterative refinement rather than absolute values.

The running times and memory requirements of a experiments on several examples are shown in Table 4. The experiments were run on a 2.6 GHz dual processor machine with 3.6 GB memory. The *Unoptimised* results are for verification using Spin. The *Iterative Refinement* contains figures obtained when using our iterative refinement procedure. The time required for model checking the abstraction obtained is shown in the *Verification* column.

The Bottling example is a more detailed version of that in Section 2. Traffic is a traffic light controller in which a large part of the system could be abstracted away. The bus protocol is a model based on the Advanced Microcontroller Bus Architecture (AMBA) from ARM [2]. The protocol supports pipelined bus transfers of different lengths with multiple masters and slaves. We verified a property about the pipelined behaviour. The abstraction that was obtained did not change the verification running time required and the refinement algorithm only introduced an overhead. The ATM example is a model of a set of ATM machines interacting with a central database. A transaction property of the

Name	Nodes	Unoptimised			Iterative Refinement			Verification
		Var	Time (s)	Memory (MB)	Var	Time (s)	Memory (MB)	Time (s)
Bottling	1	19	0.51	588.85	15	0.711	588.85	0.514
Air Conditioner	1	20	2.13	593.15	7	2.13	588.64	2.07
Traffic	2	34	12.0	2196	17	3.99	2029	3.55
Bus Protocol	1	24	3.51	2029	23	7.49	2029	3.51
ATM	2	84	> 1400	> 3207	50	53.01	2765	52.3
CD Player	2	87	—	—	26	681.2	2031	651.28
Elevator	3	98	—	—	74	> 1200	2634	107.7

Table 4: Experimental evaluation



(a) `New_drive` (b) `Old_drive`

Figure 6: Two CD drives

model could not be verified directly, but the iterative refinement algorithm could identify an abstraction that required only 52.3 seconds for verification.

The CD player is a case study of `grip 3.2.0`, an open source CD playing software [13]. We modelled the interaction between the device driver and the software as asynchronous communication between two nodes. Each widget in the graphical interface triggers a call to the device driver and was modelled as two RAs, one reacting to the user's input and the other interacting with the device driver. The RAs for the `Eject` and `Play` buttons are shown in Figure 5. The behaviour of the system varies with the kind of CD drive that is used. We partially show models of two different drives in Figure 6. In some drives, as in Figure 6(a), after an eject, the drive's state is `No_Disc` until the input *has_disc* is produced by a sensor (not shown). In contrast, in some older drives, as in Figure 6(b) (to be found on one author's laptop), the entire drive is ejected an there may still be a CD in the drive after an eject. We verified that the property that that a play command should not be issued to the drive with the tray out is not satisfied by a system with an old drive.

In the Elevator example, direct verification did not terminate. Using iterative refinement, we obtained an extremely long counterexample, which ran in the simulator for over half an hour, after which we terminated the process. We were unable to determine if the counterexample was genuine. The verification time reported is for the last abstraction computed.

6. RELATED WORK

Our verification framework is based on ideas from different areas of research. We believe this to be the first iterative-refinement algorithm for distributed reactive systems. We are unaware of similar results even for the synchronous model described here.

In a general sense, abstract interpretation is the semantic backdrop for analysis using abstractions [8]. Our use of a lattice of abstractions and the notions of minimal and best abstractions are influenced by their work.

Work in constructing modular abstractions for model checking goes back to Grumberg and Long [10]. They define a a preorder relation, similar to refinement, which is used to construct abstractions. Their model is different from ours and only synchronous parallel composition of processes is considered. More recently, Alur and Grosu [1] defined a visual language with operators including hierarchical and parallel composition. They also define refinement and prove modularity results in this model. Their model differs from the synchronous paradigm and concepts such as weak preemption are interpreted differently. Thus, the results obtained in these settings do not directly apply to our model.

A modular verification technique for synchronous systems using observers was proposed in [11]. Later, de Simone and Ressouche proposed compositional verification for Esterel [9]. The ideas in these papers do not readily fit into a counterexample guided refinement framework.

Iterative refinement appeared as localisation reduction in [12] and its recent avatar as counterexample guided abstraction-refinement in [7]. These techniques operate directly on the transition relation and do not exploit the abstraction potential of programming constructs. Similarly, Balarin's algorithm [3] for iterative-refinement of communicating finite state-machines works well for BDD based verification but is not modular and does not apply to synchronous programs.

The closest approach to ours is Vecchie and de Simone's[17] use of Esterel syntax for verification of Esterel programs. By analysing the structure of the program, they efficiently partition the BDDs representing the reachable state space of the program. Though their analysis takes program syntax into account, the transformations are eventually performed on BDDs and are at a much lower level than ours.

7. CONCLUSION

In this paper we developed a verification framework for distributed reactive systems. We define a refinement relation for such systems and show that abstractions can be constructed by dropping components from synchronous programs containing synchronous and hierarchical composition operators. We observed that the set of all such abstractions forms a semi-lattice and that elements of this lattice can be used to construct abstractions of programs that also have signal localisation and asynchronous communication. We have implemented our ideas and conducted several case studies and obtained favourable results.

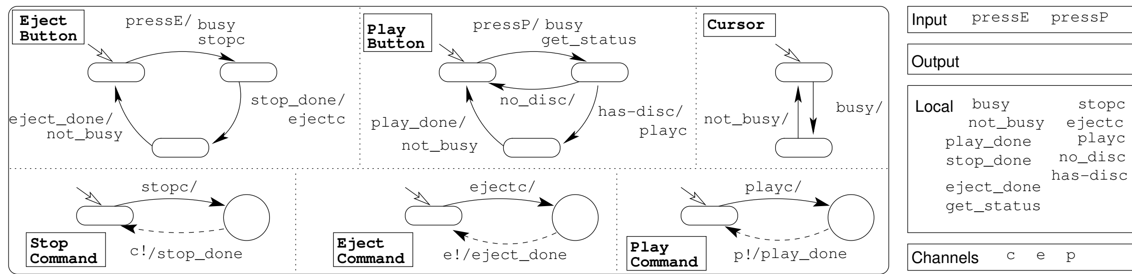


Figure 5: GUI: CD playing user interface

8. REFERENCES

[1] Rajeev Alur and Radu Grosu. Modular refinement of hierarchic reactive machines. In *Principles of Programming Languages*, pages 390–402, 2000.

[2] ARM. *Advanced Microcontroller Bus Architecture Specification*. 1999.

[3] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification*, pages 29–40, London, UK, 1993. Springer-Verlag.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91(1):64–83, January 2003.

[5] G. Berry. The constructive semantics of pure Esterel. In *Book Draft, version 3*, July 1999.

[6] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Principles of Programming Languages*, pages 85–98, 1993.

[7] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[8] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[9] Robert de Simone and Annie Ressouche. Compositional semantics of Esterel and verification by compositional reductions. In *Computer Aided Verification*, pages 441–454, 1994.

[10] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[11] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. *Algebraic Methodology and Software Technology*, pages 83–96, 1993.

[12] R. P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[13] Mike Oliphant. *Grip 3.2.0*. www.nostatic.org/grip/, 2004.

[14] S. Ramesh, A. Kulkarni, and V. Kamat. Slicing tools for synchronous reactive programs. In *Proc. of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 217–220, 2004.

[15] S. Ramesh, Sampada Sonalkar, Vijay D'Silva, N. Chandra, and B. Vijayalakshmi. A toolset for modeling and verification of GALS systems. In *Computer Aided Verification*, July 2004.

[16] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, 1997.

[17] Eric Vecchié and Robert de Simone. Syntax-driven reachable state space construction of synchronous reactive programs. In *Computer Aided Verification*, 2005.